React
入门教程

hui.liu


Published
with GitBook

目錄

React 入门教程

按照惯例，在介绍一个新技术之前总是要为它背书的，作为 React 受众在开始接触之前肯定会有一些喜闻乐见的疑问：

- 为什么不用 Backbone？
- 为什么不用 Angular？
- ...

在没有真正使用之前，其实没法评价哪一个好，没有最好的，只有最合适的，如 Why React 所说，Give it five minutes，希望你能克服初次遇到 JSX 这种存在的偏见去尝试一下。

因为官方文档组织得比较散乱，希望本教程能成为一个不错的入门参考。

有任何问题 → Github

本文档对应 React v0.14.x，使用 ES6。

React 概览

React 的核心思想是：封装组件。

各个组件维护自己的状态和 UI，当状态变更，自动重新渲染整个组件。

基于这种方式的一个直观感受就是我们不再需要不厌其烦地来回查找某个 DOM 元素，然后操作 DOM 去更改 UI。

React 大体包含下面这些概念：

- 组件
- JSX
- Virtual DOM
- Data Flow

这里通过一个简单的组件来快速了解这些概念，以及建立起对 React 的一个总体认识。

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class HelloMessage extends Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

// 加载组件到 DOM 元素 mountNode
render(<HelloMessage name="John" />, mountNode);
```

组件

React 应用都是构建在组件之上。

上面的 `HelloMessage` 就是一个 React 构建的组件，最后一句 `render` 会把这个组件显示到页面上的某个元素 `mountNode` 里面，显示的内容就是 `<div>Hello John</div>` 。

`props` 是组件包含的两个核心概念之一，另一个是 `state` （这个组件没用到）。可以把 `props` 看作是组件的配置属性，在组件内部是不变的，只是在调用这个组件的时候传入不同的属性（比如这里的 `name` ）来定制显示这个组件。

JSX

从上面的代码可以看到将 HTML 直接嵌入了 JS 代码里面，这个就是 React 提出的一种叫 *JSX* 的语法，这应该是最开始接触 React 最不能接受的设定之一，因为前端被“表现和逻辑层分离”这种思想“洗脑”太久了。但实际上组件的 HTML 是组成一个组件不可分割的一部分，能够将 HTML 封装起来才是组件的完全体，React 发明了 JSX 让 JS 支持嵌入 HTML 不得不说是一种非常聪明的做法，让前端实现真正意义上的组件化成为了可能。

好消息是你可以不一定使用这种语法，后面会进一步介绍 JSX，到时候你可能就会喜欢上了。现在要知道的是，要使用包含 JSX 的组件，是需要“编译”输出 JS 代码才能使用的，之后就会讲到开发环境。

Virtual DOM

当组件状态 `state` 有更改的时候，React 会自动调用组件的 `render` 方法重新渲染整个组件的 UI。

当然如果真的这样大面积的操作 DOM，性能会是一个很大的问题，所以 React 实现了一个*Virtual DOM*，组件 DOM 结构就是映射到这个 Virtual DOM 上，React 在这个 Virtual DOM 上实现了一个 diff 算法，当要重新渲染组件的时候，会通过 diff 寻找到要变更的 DOM 节点，再把这个修改更新到浏览器实际的 DOM 节点上，所以实际上不是真的渲染整个 DOM 树。这个 Virtual DOM 是一个纯粹的 JS 数据结构，所以性能会比原生 DOM 快很多。

Data Flow

“单向数据绑定”是 React 推崇的一种应用架构的方式。当应用足够复杂时才能体会到它的好处，虽然在一般应用场景下你可能不会意识到它的存在，也不会影响你开始使用 React，你只要先知道有这么个概念。

开发环境配置

要搭建一个现代的前端开发环境配套的工具有很多，比如 Grunt / Gulp / Webpack / Broccoli，都是要解决前端工程化问题，这个主题很大，这里为了使用 React 我们只关注其中的两个点：

- JSX 支持
- ES6 支持

好消息是业界领先的 ES6 编译工具 Babel 随着作者被 Facebook 招入麾下，已经内置了对 JSX 的支持，我们只需要配置 Babel 一个编译工具就可以了，配合 webpack 非常方便。

Webpack 配置 React 开发环境

Webpack 是一个前端资源加载/打包工具，只需要相对简单的配置就可以提供前端工程化需要的各种功能，并且如果有需要它还可以被整合到其他比如 Grunt / Gulp 的工作流。

安装 Webpack： `npm install -g webpack`

Webpack 使用一个名为 `webpack.config.js` 的配置文件，要编译 JSX，先安装对应的 loader: `npm install babel-loader --save-dev`

假设我们在当前工程目录有一个入口文件 `entry.js` ，React 组件放置在一个 `components/` 目录下，组件被 `entry.js` 引用，要使用 `entry.js` ，我们把这个文件指定输出到 `dist/bundle.js` ，Webpack 配置如下：

```
var path = require('path');

module.exports = {
    entry: './entry.js',
    output: {
        path: path.join(__dirname, '/dist'),
        filename: 'bundle.js'
    },
    resolve: {
        extensions: ['', '.js', '.jsx']
    },
    module: {
        loaders: [
            { test: /\.js|jsx$/, loaders: ['babel'] }
        ]
    }
}
```

`resolve` 指定可以被 `import` 的文件后缀。比如 `Hello.jsx` 这样的文件就可以直接用 `import Hello from 'Hello'` 引用。

`loaders` 指定 babel-loader 编译后缀名为 `.js` 或者 `.jsx` 的文件，这样你就可以在这两种类型的文件中自由使用 JSX 和 ES6 了。

监听编译: `webpack -d --watch`

更多关于 Webpack 的介绍

- webpack-howto

JSX

为什么要引入 JSX 这种语法

传统的 MVC 是将模板放在其他地方，比如 `<script>` 标签或者模板文件，再在 JS 中通过某种手段引用模板。按照这种思路，想想多少次我们面对四处分散的模板片段不知所措？纠结模板引擎，纠结模板存放位置，纠结如何引用模板……下面是一段 React 官方的看法：

> We strongly believe that components are the right way to separate concerns rather than "templates" and "display logic." We think that markup and the code that generates it are intimately tied together. Additionally, display logic is often very complex and using template languages to express it becomes cumbersome.

简单来说，React 认为组件才是王道，而组件是和模板紧密关联的，组件模板和组件逻辑分离让问题复杂化了。显而易见的道理，关键是怎么做？

所以就有了 JSX 这种语法，就是为了把 HTML 模板直接嵌入到 JS 代码里面，这样就做到了模板和组件关联，但是 JS 不支持这种包含 HTML 的语法，所以需要通过工具将 JSX 编译输出成 JS 代码才能使用。

JSX 是可选的

因为 JSX 最终是输出成 JS 代码来表达的，所以我们可以直接用 React 提供的这些 DOM 构建方法来写模板，比如一个 JSX 写的一个链接：

```
<a href="http://facebook.github.io/react/">Hello!</a>
```

用 JS 代码来写就成这样了：

```
React.createElement('a', {href: 'http://facebook.github.io/react/'}
```

你可以通过 `React.createElement` 来构造组件的 DOM 树。第一个参数是标签名，第二个参数是属性对象，第三个参数是子元素。

一个包含子元素的例子：

```
var child = React.createElement('li', null, 'Text Content');
var root = React.createElement('ul', { className: 'my-list' }, chi
React.render(root, document.body);
```

对于常见的 HTML 标签，React 已经内置了工厂方法：

```
var root = React.DOM.ul({ className: 'my-list' },
             React.DOM.li(null, 'Text Content')
           );
```

所以 JSX 和 JS 之间的转换也很简单直观，用 JSX 的好处就是它基本上就是 HTML（后面会讲到有一些小差异），对于构造 DOM 来说我们更熟悉，更具可读性。

关于 JSX 映射成 JS 对象，也就是 Virtual DOM 的内部描述，参见Virtual DOM Terminology，如果你不想使用 JSX，直接使用 JS 就是用这里面提到的接口方法。

使用 JSX

利用 JSX 编写 DOM 结构，可以用原生的 HTML 标签，也可以直接像普通标签一样引用 React 组件。这两者约定通过大小写来区分，小写的字符串是 HTML 标签，大写开头的变量是 React 组件。

使用 **HTML** 标签：

```
import React from 'react';
import { render } from 'react-dom';

var myDivElement = <div className="foo" />;
render(myDivElement, document.getElementById('mountNode'));
```

HTML 里的 `class` 在 *JSX* 里要写成 `className`，因为 `class` 在 *JS* 里是保留关键字。同理某些属性比如 `for` 要写成 `htmlFor`。

使用组件：

```
import React from 'react';
import { render } from 'react-dom';
import MyComponent from './MyComponet';

var myElement = <MyComponent someProperty={true} />;
render(myElement, document.body);
```

使用 JavaScript 表达式

属性值使用表达式，只要用 `{}` 替换 `""`:

```
// Input (JSX):
var person = <Person name={window.isLoggedIn ? window.name : ''} />
// Output (JS):
var person = React.createElement(
  Person,
  {name: window.isLoggedIn ? window.name : ''}
);
```

子组件也可以作为表达式使用：

```
// Input (JSX):
var content = <Container>{window.isLoggedIn ? <Nav /> : <Login />}<
// Output (JS):
var content = React.createElement(
  Container,
  null,
  window.isLoggedIn ? React.createElement(Nav) : React.createElemen
);
```

注释

在 JSX 里使用注释也很简单，就是沿用 JavaScript，唯一要注意的是在一个组件的子元素位置使用注释要用 `{}` 包起来。

```
var content = (
  <Nav>
      {/* child comment, put {} around */}
      <Person
        /* multi
           line
           comment */
        name={window.isLoggedIn ? window.name : ''} // end of line
      />
  </Nav>
);
```

HTML 转义

React 会将所有要显示到 DOM 的字符串转义，防止 XSS。所以如果 JSX 中含有转义后的实体字符比如 `©` (©) 最后显示到 DOM 中不会正确显示，因为 React 自动把 `©` 中的特殊字符转义了。有几种解决办法：

- 直接使用 UTF-8 字符 ©
- 使用对应字符的 Unicode 编码，查询编码
- 使用数组组装 `<div>{['cc ', <span>©</span>, ' 2015']}</div>`
- 直接插入原始的 HTML

```
<div dangerouslySetInnerHTML={{__html: 'cc &copy; 2015'}} />
```

自定义 HTML 属性

如果在 JSX 中使用的属性不存在于 HTML 的规范中，这个属性会被忽略。如果要使用自定义属性，可以用 `data-` 前缀。

可访问性属性的前缀 `aria-` 也是支持的。

支持的标签和属性

如果你要使用的某些标签或属性不在这些支持列表里面就可能被 React 忽略，必须要使用的话可以提 issue，或者用前面提到的 `dangerouslySetInnerHTML` 。

属性扩散

有时候你需要给组件设置多个属性，你不想一个个写下这些属性，或者有时候你甚至不知道这些属性的名称，这时候 *spread attributes* 的功能就很有用了。

比如：

```
var props = {};
props.foo = x;
props.bar = y;
var component = <Component {...props} />;
```

`props` 对象的属性会被设置成 `Component` 的属性。

属性也可以被覆盖：

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

写在后面的属性值会覆盖前面的属性。

关于 `...` 操作符

> The `...` operator (or spread operator) is already supported for arrays in ES6. There is also an ES7 proposal for Object Rest and Spread Properties.

JSX 与 HTML 的差异

除了前面提到的 `class` 要写成 `className` ，比较典型的还有:

- `style` 属性接受由 CSS 属性构成的 JS 对象
- `onChange` 事件表现更接近我们的直觉（不需要 onBlur 去触发）
- 表单的表现差异比较大，要单独再讲

更多异同，可以参见 DOM Differences

React 组件

可以这么说，一个 React 应用就是构建在 React 组件之上的。

组件有两个核心概念：

- props
- state

一个组件就是通过这两个属性的值在 `render` 方法里面生成这个组件对应的 HTML 结构。

注意：组件生成的 *HTML* 结构只能有一个单一的根节点。

props

前面也提到很多次了，`props` 就是组件的属性，由外部通过 JSX 属性传入设置，一旦初始设置完成，就可以认为 `this.props` 是不可更改的，所以不要轻易更改设置 `this.props` 里面的值（虽然对于一个 JS 对象你可以做任何事）。

state

`state` 是组件的当前状态，可以把组件简单看成一个“状态机”，根据状态 `state` 呈现不同的 UI 展示。

一旦状态（数据）更改，组件就会自动调用 `render` 重新渲染 UI，这个更改的动作会通过 `this.setState` 方法来触发。

划分状态数据

一条原则：让组件尽可能地少状态。

这样组件逻辑就越容易维护。

什么样的数据属性可以当作状态？

当更改这个状态（数据）需要更新组件 UI 的就可以认为是 `state` ，下面这些可以认为不是状态：

- 可计算的数据：比如一个数组的长度
- 和 props 重复的数据：除非这个数据是要做变更的

最后回过头来反复看几遍 Thinking in React，相信会对组件有更深刻的认识。

无状态组件

你也可以用纯粹的函数来定义无状态的组件(stateless function)，这种组件没有状态，没有生命周期，只是简单的接受 props 渲染生成 DOM 结构。无状态组件非常简单，开销很低，如果可能的话尽量使用无状态组件。比如使用箭头函数定义：

```
const HelloMessage = (props) => <div> Hello {props.name}</div>;
render(<HelloMessage name="John" />, mountNode);
```

因为无状态组件只是函数，所以它没有实例返回，这点在想用 refs 获取无状态组件的时候要注意，参见DOM 操作。

组件生命周期

一般来说，一个组件类由 `extends Component` 创建，并且提供一个 `render` 方法以及其他可选的生命周期函数、组件相关的事件或方法来定义。

一个简单的例子：

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class LikeButton extends Component {
  constructor(props) {
    super(props);
    this.state = { liked: false };
  }

  handleClick(e) {
    this.setState({ liked: !this.state.liked });
  }

  render() {
    const text = this.state.liked ? 'like' : 'haven\'t liked';
    return (
      <p onClick={this.handleClick.bind(this)}>
          You {text} this. Click to toggle.
      </p>
    );
  }
}

render(
    <LikeButton />,
    document.getElementById('example')
);
```

`getInitialState`

初始化 `this.state` 的值，只在组件装载之前调用一次。

如果是使用 ES6 的语法，你也可以在构造函数中初始化状态，比如：

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: props.initialCount };
  }

  render() {
    // ...
  }
}
```

`getDefaultProps`

只在组件创建时调用一次并缓存返回的对象（即在 `React.createClass` 之后就会调用）。

因为这个方法在实例初始化之前调用，所以在这个方法里面不能依赖 `this` 获取到这个组件的实例。

在组件装载之后，这个方法缓存的结果会用来保证访问 `this.props` 的属性时，当这个属性没有在父组件中传入（在这个组件的 JSX 属性里设置），也总是有值的。

如果是使用 ES6 语法，可以直接定义 `defaultProps` 这个类属性来替代，这样能更直观的知道 default props 是预先定义好的对象值：

```
Counter.defaultProps = { initialCount: 0 };
```

`render`

必须

组装生成这个组件的 HTML 结构（使用原生 HTML 标签或者子组件），也可以返回 `null` 或者 `false`，这时候 `ReactDOM.findDOMNode(this)` 会返回 `null`。

生命周期函数

装载组件触发

`componentWillMount`

只会在装载之前调用一次，在 `render` 之前调用，你可以在这个方法里面调用 `setState` 改变状态，并且不会导致额外调用一次 `render`

`componentDidMount`

只会在装载完成之后调用一次，在 `render` 之后调用，从这里开始可以通过 `ReactDOM.findDOMNode(this)` 获取到组件的 DOM 节点。

更新组件触发

这些方法不会在首次 `render` 组件的周期调用

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `componentDidUpdate`

卸载组件触发

- `componentWillUnmount`

更多关于组件相关的方法说明，参见：

- Component Specs
- Component Lifecycle
- Component API

事件处理

一个简单的例子：

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class LikeButton extends Component {
  constructor(props) {
    super(props);
    this.state = { liked: false };
  }

  handleClick(e) {
    this.setState({ liked: !this.state.liked });
  }

  render() {
    const text = this.state.liked ? 'like' : 'haven\'t liked';
    return (
      <p onClick={this.handleClick.bind(this)}>
          You {text} this. Click to toggle.
      </p>
    );
  }
}

render(
    <LikeButton />,
    document.getElementById('example')
);
```

可以看到 React 里面绑定事件的方式和在 HTML 中绑定事件类似，使用驼峰式命名指定要绑定的 `onClick` 属性为组件定义的一个方法 `{this.handleClick.bind(this)}` 。

注意要显式调用 `bind(this)` 将事件函数上下文绑定要组件实例上，这也是 React 推崇的原则：没有黑科技，尽量使用显式的容易理解的 JavaScript 代码。

“合成事件”和“原生事件”

React 实现了一个“合成事件”层（synthetic event system），这个事件模型保证了和 W3C 标准保持一致，所以不用担心有什么诡异的用法，并且这个事件层消除了 IE 与 W3C 标准实现之间的兼容问题。

“合成事件”还提供了额外的好处：

事件委托

“合成事件”会以事件委托（event delegation）的方式绑定到组件最上层，并且在组件卸载（unmount）的时候自动销毁绑定的事件。

什么是“原生事件”？

比如你在 `componentDidMount` 方法里面通过 `addEventListener` 绑定的事件就是浏览器原生事件。

使用原生事件的时候注意在 `componentWillUnmount` 解除绑定 `removeEventListener` 。

所有通过 JSX 这种方式绑定的事件都是绑定到“合成事件”，除非你有特别的理由，建议总是用 React 的方式处理事件。

Tips

关于这两种事件绑定的使用，这里有必要分享一些额外的人生经验

如果混用“合成事件”和“原生事件”，比如一种常见的场景是用原生事件在 document 上绑定，然后在组件里面绑定的合成事件想要通过 `e.stopPropagation()` 来阻止事件冒泡到 document，这时候是行不通的，参见 Event delegation，因为 `e.stopPropagation` 是内部“合成事件” 层面的，解决方法是要用 `e.nativeEvent.stopImmediatePropagation()`

”合成事件“ 的 `event` 对象只在当前 event loop 有效，比如你想在事件里面调用一个 promise，在 resolve 之后去拿 `event` 对象会拿不到（并且没有错误抛出）：

```
handleClick(e) {
  promise.then(() => doSomethingWith(e));
}
```

详情见 Event pooling 说明。

参数传递

给事件处理函数传递额外参数的方式：`bind(this, arg1, arg2, ...)`

```
render: function() {
    return <p onClick={this.handleClick.bind(this, 'extra param')}>
},
handleClick: function(param, event) {
    // handle click
}
```

React 支持的事件列表

DOM 操作

大部分情况下你不需要通过查询 DOM 元素去更新组件的 UI，你只要关注设置组件的状态（ `setState` ）。但是可能在某些情况下你确实需要直接操作 DOM。

首先我们要了解 `ReactDOM.render` 组件返回的是什么？

它会返回对组件的引用也就是组件实例（对于无状态状态组件来说返回 null），注意 JSX 返回的不是组件实例，它只是一个 `ReactElement` 对象（还记得我们用纯 JS 来构建 JSX 的方式吗），比如这种：

```
// A ReactElement
const myComponent = <MyComponent />

// render
const myComponentInstance = ReactDOM.render(myComponent, mountNode)
myComponentInstance.doSomething();
```

findDOMNode()

当组件加载到页面上之后（mounted），你都可以通过 `react-dom` 提供的 `findDOMNode()` 方法拿到组件对应的 DOM 元素。

```
import { findDOMNode } from 'react-dom';

// Inside Component class
componentDidMound() {
  const el = findDOMNode(this);
}
```

`findDOMNode()` 不能用在无状态组件上。

Refs

另外一种方式就是通过在要引用的 DOM 元素上面设置一个 `ref` 属性指定一个名称，然后通过 `this.refs.name` 来访问对应的 DOM 元素。

比如有一种情况是必须直接操作 DOM 来实现的，你希望一个 `<input/>` 元素在你清空它的值时 focus，你没法仅仅靠 `state` 来实现这个功能。

```
class App extends Component {
  constructor() {
    return { userInput: '' };
  }

  handleChange(e) {
    this.setState({ userInput: e.target.value });
  }

  clearAndFocusInput() {
    this.setState({ userInput: '' }, () => {
      this.refs.theInput.focus();
    });
  }

  render() {
    return (
      <div>
        <div onClick={this.clearAndFocusInput.bind(this)}>
          Click to Focus and Reset
        </div>
        <input
          ref="theInput"
          value={this.state.userInput}
          onChange={this.handleChange.bind(this)}
        />
      </div>
    );
  }
}
```

如果 `ref` 是设置在原生 HTML 元素上，它拿到的就是 DOM 元素，如果设置在自定义组件上，它拿到的就是组件实例，这时候就需要通过 `findDOMNode` 来拿到组件的 DOM 元素。

因为无状态组件没有实例，所以 ref 不能设置在无状态组件上，一般来说这没什么问题，因为无状态组件没有实例方法，不需要 ref 去拿实例调用相关的方法，但是如果想要拿无状态组件的 DOM 元素的时候，就需要用一个状态组件封装一层，然后通过 `ref` 和 `findDOMNode` 去获取。

总结

- 你可以使用 ref 到的组件定义的任何公共方法，比如 `this.refs.myTypeahead.reset()`
- Refs 是访问到组件内部 DOM 节点唯一可靠的方法
- Refs 会自动销毁对子组件的引用（当子组件删除时）

注意事项

- 不要在 `render` 或者 `render` 之前访问 `refs`
- 不要滥用 `refs`，比如只是用它来按照传统的方式操作界面 UI：找到 DOM -> 更新 DOM

组合组件

使用组件的目的就是通过构建模块化的组件，相互组合组件最后组装成一个复杂的应用。

在 React 组件中要包含其他组件作为子组件，只需要把组件当作一个 DOM 元素引入就可以了。

一个例子：一个显示用户头像的组件 `Avatar` 包含两个子组件 `ProfilePic` 显示用户头像和 `ProfileLink` 显示用户链接：

```
import React from 'react';
import { render } from 'react-dom';

const ProfilePic = (props) => {
  return (
    <img src={'http://graph.facebook.com/' + props.username + '/pic
  );
}

const ProfileLink = (props) => {
  return (
    <a href={'http://www.facebook.com/' + props.username}>
      {props.username}
    </a>
  );
}

const Avatar = (props) => {
  return (
    <div>
      <ProfilePic username={props.username} />
      <ProfileLink username={props.username} />
    </div>
  );
}

render(
  <Avatar username="pwh" />,
  document.getElementById('example')
);
```

通过 `props` 传递值。

循环插入子元素

如果组件中包含通过循环插入的子元素，为了保证重新渲染 UI 的时候能够正确显示这些子元素，每个元素都需要通过一个特殊的 `key` 属性指定一个唯一值。具体原因见这里，为了内部 diff 的效率。

`key` 必须直接在循环中设置：

```
const ListItemWrapper = (props) => <li>{props.data.text}</li>;

const MyComponent = (props) => {
  return (
    <ul>
      {props.results.map((result) => {
        return <ListItemWrapper key={result.id} data={result}/>;
      })}
    </ul>
  );
}
```

你也可以用一个 `key` 值作为属性，子元素作为属性值的对象字面量来显示子元素列表，虽然这种用法的场景有限，参见Keyed Fragments，但是在这种情况下要注意生成的子元素重新渲染后在 DOM 中显示的顺序问题。

实际上浏览器在遍历一个字面量对象的时候会保持顺序一致，除非存在属性值可以被转换成整数值，这种属性值会排序并放在其他属性之前被遍历到，所以为了防止这种情况发生，可以在构建这个字面量的时候在 `key` 值前面加字符串前缀，比如：

```
render() {
  var items = {};

  this.props.results.forEach((result) => {
    // If result.id can look like a number (consider short hashes),
    // object iteration order is not guaranteed. In this case, we a
    // to ensure the keys are strings.
    items['result-' + result.id] = <li>{result.text}</li>;
  });

  return (
    <ol>
      {items}
    </ol>
   );
}
```

`this.props.children`

组件标签里面包含的子元素会通过 `props.children` 传递进来。

比如：

```
React.render(<Parent><Child /></Parent>, document.body);

React.render(<Parent><span>hello</span>{'world'}</Parent>, document
```

HTML 元素会作为 React 组件对象、JS 表达式结果是一个文字节点，都会存入 `Parent` 组件的 `props.children` 。

一般来说，可以直接将这个属性作为父组件的子元素 render：

```
const Parent = (props) => <div>{props.children}</div>;
```

`props.children` 通常是一个组件对象的数组，但是当只有一个子元素的时候， `props.children` 将是这个唯一的子元素，而不是数组了。

`React.Children` 提供了额外的方法方便操作这个属性。

组件间通信

父子组件间通信

这种情况下很简单，就是通过 `props` 属性传递，在父组件给子组件设置 `props`，然后子组件就可以通过 `props` 访问到父组件的数据／方法，这样就搭建起了父子组件间通信的桥梁。

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class GroceryList extends Component {
  handleClick(i) {
    console.log('You clicked: ' + this.props.items[i]);
  }

  render() {
    return (
      <div>
        {this.props.items.map((item, i) => {
          return (
            <div onClick={this.handleClick.bind(this, i)} key={i}><
          );
        })}
      </div>
    );
  }
}

render(
  <GroceryList items={['Apple', 'Banana', 'Cranberry']} />, mountNo
);
```

`div` 可以看作一个子组件，指定它的 `onClick` 事件调用父组件的方法。

父组件访问子组件？用 `refs`

非父子组件间的通信

使用全局事件 Pub/Sub 模式，在 `componentDidMount` 里面订阅事件，在 `componentWillUnmount` 里面取消订阅，当收到事件触发的时候调用 `setState` 更新 UI。

这种模式在复杂的系统里面可能会变得难以维护，所以看个人权衡是否将组件封装到大的组件，甚至整个页面或者应用就封装到一个组件。

一般来说，对于比较复杂的应用，推荐使用类似 Flux 这种单项数据流架构，参见 Data Flow。

Mixins

NOTE: 使用 ES6 class 定义的组件已经不支持 mixin 了，因为使用 mixin 的场景都可以用组合组件这种模式来做到，参见 Mixins Are Dead. Long Live Composition

这里暂时留存这部分内容。

虽然组件的原则就是模块化，彼此之间相互独立，但是有时候不同的组件之间可能会共用一些功能，共享一部分代码。所以 React 提供了 `mixins` 这种方式来处理这种问题。

Mixin 就是用来定义一些方法，使用这个 mixin 的组件能够自由的使用这些方法（就像在组件中定义的一样），所以 mixin 相当于组件的一个扩展，在 mixin 中也能定义“生命周期”方法。

比如一个定时器的 mixin：

```
var SetIntervalMixin = {
    componentWillMount: function() {
        this.intervals = [];
    },
    setInterval: function() {
        this.intervals.push(setInterval.apply(null, arguments));
    },
    componentWillUnmount: function() {
        this.intervals.map(clearInterval);
    }
};

var TickTock = React.createClass({
    mixins: [SetIntervalMixin], // Use the mixin
    getInitialState: function() {
        return {seconds: 0};
    },
    componentDidMount: function() {
        this.setInterval(this.tick, 1000); // Call a method on the
    },
    tick: function() {
        this.setState({seconds: this.state.seconds + 1});
    },
    render: function() {
        return (
            <p>
                React has been running for {this.state.seconds} sec
            </p>
        );
    }
});

React.render(
    <TickTock />,
    document.getElementById('example')
);
```

React 的 `mixins` 的强大之处在于，如果一个组件使用了多个 mixins，其中几个 `mixins` 定义了相同的“生命周期方法”，这些方法会在组件相应的方法执行完之后按 mixins 指定的数组顺序执行。

Data Flow

Data Flow 只是一种应用架构的方式，比如数据如何存放，如何更改数据，如何通知数据更改等等，所以它不是 React 提供的额外的什么新功能，可以看成是使用 React 构建大型应用的一种最佳实践。

正因为它是这样一种概念，所以涌现了许多实现，这里主要关注两种实现：

- 官方的 Flux
- 更优雅的 Redux

Flux

React 标榜自己是 MVC 里面 V 的部分，那么 Flux 就相当于添加 M 和 C 的部分。

Flux 是 Facebook 使用的一套前端应用的架构模式。

一个 Flux 应用主要包含四个部分：

- the dispatcher

  处理动作分发，维护 Store 之间的依赖关系

- the stores

  数据和逻辑部分

- the views

  React 组件，这一层可以看作 controller-views，作为视图同时响应用户交互

- the actions

  提供给 dispatcher 传递数据给 store

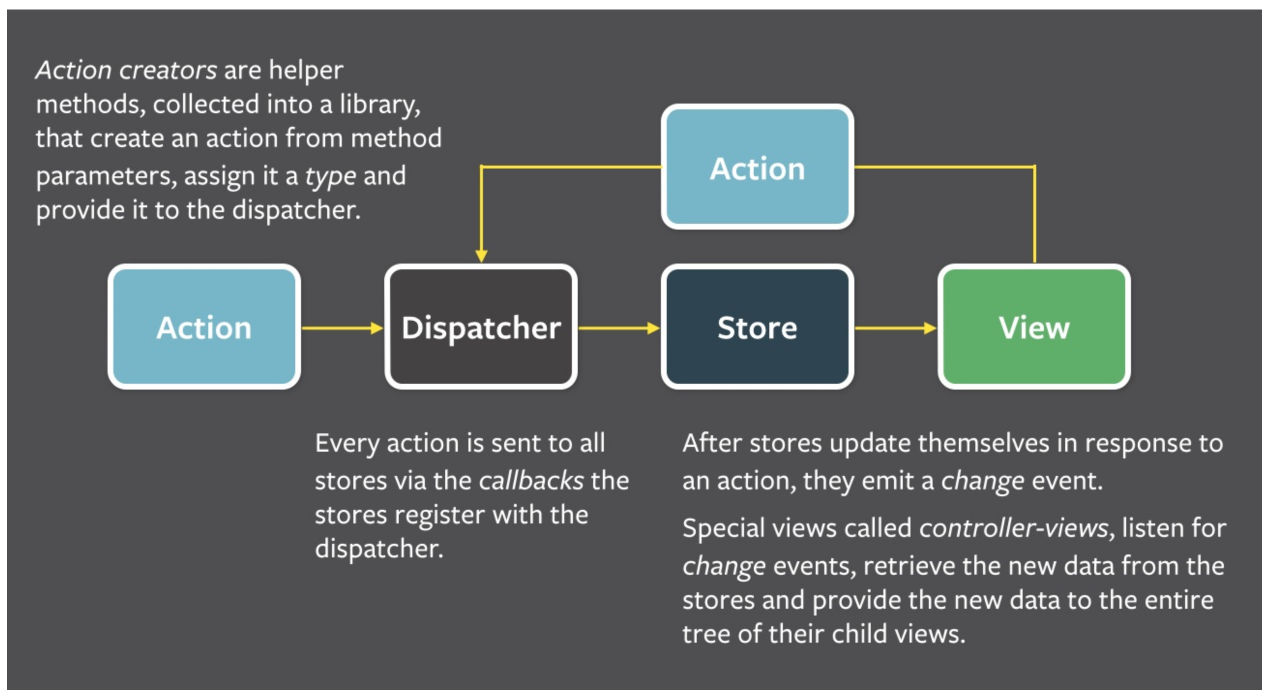
针对上面提到的 Flux 这些概念，需要写一个简单的类库来实现衔接这些功能，市面上有很多种实现，这里讨论 Facebook 官方的一个实现 Dispatcher.js

单向数据流

先来了解一下 Flux 的核心"单向数据流"怎么运作的：

```
Action -> Dispatcher -> Store -> View
```

更多时候 View 会通过用户交互触发 Action，所以一个简单完整的数据流类似这样：



整个流程如下：

- 首先要有 action，通过定义一些 *action creator* 方法根据需要创建 Action 提供给 dispatcher
- View 层通过用户交互（比如 onClick）会触发 Action
- Dispatcher 会分发触发的 Action 给所有注册的 Store 的回调函数
- Store 回调函数根据接收的 Action 更新自身数据之后会触发一个 *change* 事件通知 View 数据更改了
- View 会监听这个 *change* 事件，拿到对应的新数据并调用 `setState` 更新组件 UI

所有的状态都由 Store 来维护，通过 Action 传递数据，构成了如上所述的单向数据流循环，所以应用中的各部分分工就相当明确，高度解耦了。

这种单向数据流使得整个系统都是透明可预测的。

Dispatcher

一个应用只需要一个 dispatcher 作为分发中心，管理所有数据流向，分发动作给 Store，没有太多其他的逻辑（一些 *action creator* 方法也可以放到这里）。

Dispatcher 分发动作给 Store 注册的回调函数，这和一般的订阅/发布模式不同的地方在于：

- 回调函数不是订阅到某一个特定的事件/频道，每个动作会分发给所有注册的回

调函数
- 回调函数可以指定在其他回调之后调用

基于 Flux 的架构思路，Dispatcher.js 提供的 API 很简单：

- **register(function callback): string** 注册回调函数，返回一个 token 供在 `waitFor()` 使用
- **unregister(string id): void** 通过 token 移除回调
- **waitFor(array ids): void** 在指定的回调函数执行之后才执行当前回调。这个方法只能在分发动作的回调函数中使用
- **dispatch(object payload): void** 分发动作 payload 给所有注册回调
- **isDispatching(): boolean** 返回 Dispatcher 当前是否处在分发的状态

dispatcher 只是一个粘合剂，剩余的 Store、View、Action 就需要按具体需求去实现了。

接下来结合 flux-todomvc 这个简单的例子，提取其中的关键部分，看一下实际应用中如何衔接 Flux 整个流程，希望能对 Flux 各个部分有更直观深入的理解。

Action

首先要创建动作，通过定义一些 *action creator* 方法来创建，这些方法用来暴露给外部调用，通过 `dispatch` 分发对应的动作，所以 *action creator* 也称作 *dispatcher helper methods* 辅助 dipatcher 分发。 参见 actions/TodoActions.js

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var TodoConstants = require('../constants/TodoConstants');

var TodoActions = {
  create: function(text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },

  updateText: function(id, text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_UPDATE_TEXT,
      id: id,
      text: text
    });
  },

  // 不带 payload 数据的动作
  toggleCompleteAll: function() {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_TOGGLE_COMPLETE_ALL
    });
  }
};
```

`AppDispatcher` 直接继承自 Dispatcher.js，在这个简单的例子中没有提供什么额外的功能。 `TodoConstants` 定义了动作的类型名称常量。

类似 `create` 、 `updateText` 就是 *action creator*，这两个动作会通过 View 上的用户交互触发（比如输入框）。 除了用户交互会创建动作，服务端接口调用也可以用来创建动作，比如通过 Ajax 请求的一些初始数据也可以创建动作提供给 dispatcher，再分发给 store 使用这些初始数据。

> action creators are nothing more than a call into the dispatcher.

可以看到所谓动作就是用来封装传递数据的，动作只是一个简单的对象，包含两部分：payload（数据）和 type（类型），type 是一个字符串常量，用来标识动作。

Store

Stores 包含应用的状态和逻辑，不同的 Store 管理应用中不同部分的状态。如 stores/TodoStore.js

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var TodoConstants = require('../constants/TodoConstants');
var assign = require('object-assign');

var CHANGE_EVENT = 'change';

var _todos = {};

// 先定义一些数据处理方法
function create(text) {
  var id = (+new Date() + Math.floor(Math.random() * 999999)).toSt
  _todos[id] = {
    id: id,
    complete: false,
    text: text
  };
}
function update(id, updates) {
  _todos[id] = assign({}, _todos[id], updates);
}
// ...

var TodoStore = assign({}, EventEmitter.prototype, {
  // Getter 方法暴露给外部获取 Store 数据
  getAll: function() {
    return _todos;
  },
  // 触发 change 事件
  emitChange: function() {
    this.emit(CHANGE_EVENT);
  },
  // 提供给外部 View 绑定 change 事件
  addChangeListener: function(callback) {
```

```
      this.on(CHANGE_EVENT, callback);
    }
  });

  // 注册到 dispatcher，通过动作类型过滤处理当前 Store 关心的动作
  AppDispatcher.register(function(action) {
    var text;

    switch(action.actionType) {
      case TodoConstants.TODO_CREATE:
        text = action.text.trim();
        if (text !== '') {
          create(text);
        }
        TodoStore.emitChange();
        break;

      case TodoConstants.TODO_UPDATE_TEXT:
        text = action.text.trim();
        if (text !== '') {
          update(action.id, {text: text});
        }
        TodoStore.emitChange();
        break;
    }
  });
```

在 Store 注册给 dispatcher 的回调函数中会接受到分发的 action，因为每个 action 都会分发给所有注册的回调，所以回调函数里面要判断这个 action 的 *type* 并调用相关的内部方法处理更新 action 带过来的数据（payload），再通知 view 数据变更。

Store 里面不会暴露直接操作数据的方法给外部，暴露给外部调用的方法都是 Getter 方法，没有 Setter 方法，唯一更新数据的手段就是通过在 dispatcher 注册的回调函数。

View

View 就是 React 组件，从 Store 获取状态（数据），绑定 change 事件处理。如 components/TodoApp.react.js

```
var React = require('react');
var TodoStore = require('../stores/TodoStore');

function getTodoState() {
  return {
    allTodos: TodoStore.getAll(),
    areAllComplete: TodoStore.areAllComplete()
  };
}

var TodoApp = React.createClass({

  getInitialState: function() {
    return getTodoState();
  },

  componentDidMount: function() {
    TodoStore.addChangeListener(this._onChange);
  },

  componentWillUnmount: function() {
    TodoStore.removeChangeListener(this._onChange);
  },

  render: function() {
    return <div>/*...*/</div>
  },

  _onChange: function() {
    this.setState(getTodoState());
  }
});
```

一个 View 可能关联多个 Store 来管理不同部分的状态，得益于 React 更新 View 如此简单（ `setState` ），复杂的逻辑都被 Store 隔离了。

更多资料

- Flux chat 很简洁明了的一个 Slide
- flux-chat source code 一个更复杂一点的例子

Redux

Dan Abramov 在 React Europe 2015 上作了一场令人印象深刻的演示 Hot Reloading with Time Travel，之后 Redux 迅速成为最受人关注的 Flux 实现之一。

Redux 把自己标榜为一个“可预测的状态容器”，其实也是 Flux 里面“单向数据流”的思想，只是它充分利用函数式的特性，让整个实现更加优雅纯粹，使用起来也更简单。

```
Redux(oldState) => newState
```

Redux 是超越 Flux 的一次进化。

进化 Flux

我们可以先通过对比 Redux 和 Flux 的实现来感受一下 Redux 带来的惊艳。

首先是 *action creators*，Flux 是直接在 action 里面调用 dispatch：

```
export function addTodo(text) {
  AppDispatcher.dispatch({
    type: ActionTypes.ADD_TODO,
    text: text
  });
}
```

Redux 把它简化成了这样：

```
export function addTodo(text) {
  return {
    type: ActionTypes.ADD_TODO,
    text: text
  };
}
```

这一步把 dispatcher 和 action 解藕了，很快我们就能看到它带来的好处。

接下来是 Store，这是 Flux 里面的 Store：

```
let _todos = [];
const TodoStore = Object.assign(new EventEmitter(), {
  getTodos() {
    return _todos;
  }
});
AppDispatcher.register(function (action) {
  switch (action.type) {
  case ActionTypes.ADD_TODO:
    _todos = _todos.concat([action.text]);
    TodoStore.emitChange();
    break;
  }
});
export default TodoStore;
```

Redux 把它简化成了这样：

```
const initialState = { todos: [] };
export default function TodoStore(state = initialState, action) {
  switch (action.type) {
  case ActionTypes.ADD_TODO:
    return { todos: state.todos.concat([action.text]) };
  default:
    return state;
}
```

同样把 dispatch 从 Store 里面剥离了，Store 变成了一个 **pure function**： `(state, action) => state`

什么是 pure function

如果一个函数没有任何副作用（side-effects)，不会影响任何外部状态，对于任何一个相同的输入（参数），无论何时调用这个函数总是返回同样的结果，这个函数就是一个 pure function。所谓 side-effects 就是会改变外部状态的因素 ，比如 Ajax 请求就有 side-effects，因为它带来了不确定性。

所以现在 Store 不再拥有状态，而只是管理状态，所以首先要明确一个概念，Store 和 State 是有区别的，Store 并不是一个简单的数据结构，State 才是，Store 会包含一些方法来管理 State，比如获取／修改 State。

基于这样的 Store，可以做很多扩展，这也是 Redux 强大之处。

来源：The Evolution of Flux Frameworks

Redux 的基础概念

三个基本原则

- 整个应用只有唯一一个可信数据源，也就是只有一个 Store
- State 只能通过触发 Action 来更改
- State 的更改必须写成纯函数，也就是每次更改总是返回一个新的 State，在 Redux 里这种函数称为 Reducer

Actions

Action 很简单，就是一个单纯的包含 `{ type, payload }` 的对象，`type` 是一个常量用来标示动作类型，`payload` 是这个动作携带的数据。Action 需要通过 `store.dispatch()` 方法来发送。

比如一个最简单的 action：

```
{
  type: 'ADD_TODO',
  text: 'Build my first Redux app'
}
```

一般来说，会使用函数（Action Creators）来生成 action，这样会有更大的灵活性，Action Creators 是一个 **pure function**，它最后会返回一个 action 对象：

```
function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

所以现在要触发一个动作只要调用 `dispatch`：`dispatch(addTodo(text))`

稍后会讲到如何拿到 `store.dispatch`

Reducers

Reducer 用来处理 Action 触发的对状态树的更改。

所以一个 reducer 函数会接受 `oldState` 和 `action` 两个参数，返回一个新的 state：`(oldState, action) => newState`。一个简单的 reducer 可能类似这样：

```
const initialState = {
  a: 'a',
  b: 'b'
};

function someApp(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_A':
      return { ...state, a: 'Modified a' };
    case 'CHANGE_B':
      return { ...state, b: action.payload };
    default:
      return state
  }
}
```

值得注意的有两点：

- 我们用到了 object spread 语法 确保不会更改到 `oldState` 而是返回一个 `newState`
- 对于不需要处理的 action，直接返回 `oldState`

Reducer 也是 **pure function**，这点非常重要，所以绝对不要在 reducer 里面做一些引入 side-effects 的事情，比如：

- 直接修改 state 参数对象
- 请求 API
- 调用不纯的函数，比如 `Data.now()` `Math.random()`

因为 Redux 里面只有一个 Store，对应一个 State 状态，所以整个 State 对象就是由一个 reducer 函数管理，但是如果所有的状态更改逻辑都放在这一个 reducer 里面，显然会变得越来越巨大，越来越难以维护。得益于纯函数的实现，我们只需要稍微变通一下，让状态树上的每个字段都有一个 reducer 函数来管理就可以拆分成很小的 reducer 了：

```
function someApp(state = {}, action) {
  return {
    a: reducerA(state.a, action),
    b: reducerB(state.b, action)
  };
}
```

对于 `reducerA` 和 `reducerB` 来说，他们依然是形如：`(oldState, action) => newState` 的函数，只是这时候的 state 不是整个状态树，而是树上的特定字段，每个 reducer 只需要判断 action，管理自己关心的状态字段数据就好了。

Redux 提供了一个工具函数 `combineReducers` 来简化这种 reducer 合并：

```
import { combineReducers } from 'redux';

const someApp = combineReducers({
  a: reducerA,
  b: reducerB
});
```

如果 reducer 函数名字和字段名字相同，利用 ES6 的 Destructuring 可以进一步简化成：`combineReducers({ a, b })`

象 `someApp` 这种管理整个 State 的 reducer，可以称为 **root reducer**。

Store

现在有了 Action 和 Reducer，Store 的作用就是连接这两者，Store 的作用有这么几个：

- Hold 住整个应用的 State 状态树

- 提供一个 `getState()` 方法获取 State
- 提供一个 `dispatch()` 方法发送 action 更改 State
- 提供一个 `subscribe()` 方法注册回调函数监听 State 的更改

创建一个 Store 很容易，将 **root reducer** 函数传递给 `createStore` 方法即可：

```
import { createStore } from 'redux';
import someApp from './reducers';
let store = createStore(someApp);

// 你也可以额外指定一个初始 State（initialState），这对于服务端渲染很有用
// let store = createStore(someApp, window.STATE_FROM_SERVER);
```

现在我们就拿到了 `store.dispatch`，可以用来分发 action 了：

```
let unsubscribe = store.subscribe(() => console.log(store.getState(

// Dispatch
store.dispatch({ type: 'CHANGE_A' });
store.dispatch({ type: 'CHANGE_B', payload: 'Modified b' });

// Stop listening to state updates
unsubscribe();
```

Data Flow

以上提到的 `store.dispatch(action) -> reducer(state, action) -> store.getState()` 其实就构成了一个“单向数据流”，我们再来总结一下。

1. 调用 `store.dispatch(action)`

Action 是一个包含 `{ type, payload }` 的对象，它描述了“发生了什么”，比如：

```
{ type: 'LIKE_ARTICLE', articleID: 42 }
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }
{ type: 'ADD_TODO', text: 'Read the Redux docs.' }
```

你可以在任何地方调用 `store.dispatch(action)` ，比如组件内部，Ajax 回调函数里面等等。

2. Action 会触发给 Store 指定的 root reducer

root reducer 会返回一个完整的状态树，State 对象上的各个字段值可以由各自的 reducer 函数处理并返回新的值。

- reducer 函数接受 `(state, action)` 两个参数
- reducer 函数判断 `action.type` 然后处理对应的 `action.payload` 数据来更新并返回一个新的 state

3. Store 会保存 root reducer 返回的状态树

新的 State 会替代旧的 State，然后所有 `store.subscribe(listener)` 注册的回调函数会被调用，在回调函数里面可以通过 `store.getState()` 拿到新的 State。

这就是 Redux 的运作流程，接下来看如何在 React 里面使用 Redux。

在 React 应用中使用 Redux

和 Flux 类似，Redux 也是需要注册一个回调函数 `store.subscribe(listener)` 来获取 State 的更新，然后我们要在 `listener` 里面调用 `setState()` 来更新 React 组件。

Redux 官方提供了 react-redux 来简化 React 和 Redux 之间的绑定，不再需要像 Flux 那样手动注册／解绑回调函数。

接下来看一下是怎么做到的，react-redux 只有两个 API

<Provider>

`<Provider>` 作为一个容器组件，用来接受 Store，并且让 Store 对子组件可用，用法如下：

```
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import App from './app';

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

这时候 `<Provider>` 里面的子组件 `<App />` 才可以使用 `connect` 方法关联 store。

`<Provider>` 的实现很简单，他利用了 React 一个（暂时）隐藏的特性 `Contexts`，`Context` 用来传递一些父容器的属性对所有子孙组件可见，在某些场景下面避免了用 `props` 传递多层组件的繁琐，要想更详细了解 `Contexts` 可以参考这篇文章。

Connect

`connect()` 这个方法略微复杂一点，主要是因为它的用法非常灵活： `connect([mapStateToProps], mapDispatchToProps], [mergeProps], [options])` ，它最多接受4个参数，都是可选的，并且这个方法调用会返回另一个函数，这个返回的函数来接受一个组件类作为参数，最后才返回一个和 Redux store 关联起来的新组件，类似这样：

```
class App extends Component { ... }

export default connect()(App);
```

这样就可以在 `App` 这个组件里面通过 `props` 拿到 Store 的 `dispatch` 方法，但是注意现在的 `App` 没有监听 Store 的状态更改，如果要监听 Store 的状态更改，必须要指定 `mapStateToProps` 参数。

先来看它的参数：

- `[mapStateToProps(state, [ownProps]): stateProps]` ：第一个可选参数是一个函数，只有指定了这个参数，这个关联（connected）组件才会监听 Redux Store 的更新，每次更新都会调用 `mapStateToProps` 这个函数，返回一个字面量对象将会合并到组件的 `props` 属性。 `ownProps` 是可选的第二个参数，它是传递给组件的 `props` ，当组件获取到新的 `props` 时， `ownProps` 都会拿到这个值并且执行 `mapStateToProps` 这个函数。
- `[mapDispatchProps(dispatch, [ownProps]): dispatchProps]` ：这个函数用来指定如何传递 `dispatch` 给组件，在这个函数里面直接 dispatch action creator，返回一个字面量对象将会合并到组件的 `props` 属性，这样关联组件可以直接通过 `props` 调用到 `action` ， Redux 提供了一个 `bindActionCreators()` 辅助函数来简化这种写法。 如果省略这个参数，默认直接把 `dispatch` 作为 `props` 传入。 `ownProps` 作用同上。

剩下的两个参数比较少用到，更详细的说明参看官方文档，其中提供了很多简单清晰的用法示例来说明这些参数。

一个具体一点的例子

Redux 创建 Store，Action，Reducer 这部分就省略了，这里只看 react-redux 的部分。

```
import React, { Component } from 'react';
import someActionCreator from './actions/someAction';
import * as actionCreators from './actions/otherAction';

function mapStateToProps(state) {
  return {
    propName: state.propName
  };
}

function mapDispatchProps(dispatch) {
  return {
    someAction: (arg) => dispatch(someActionCreator(arg)),
    otherActions: bindActionCreators(actionCreators, dispatch)
  };
}

class App extends Component {
  render() {
    // `mapStateToProps` 和 `mapDispatchProps` 返回的字段都是 `props`
    const { propName, someAction, otherActions } = this.props;
    return (
      <div onClick={someAction.bind(this, 'arg')}>
        {propName}
      </div>
    );
  }
}

export default connect(mapStateToProps, mapDispatchProps)(App);
```

如前所述，这个 connected 的组件必须放到 `<Provider>` 的容器里面，当 State 更改的时候就会自动调用 `mapStateToProps` 和 `mapDispatchProps` 从而更新组件的 `props`。组件内部也可以通过 `props` 调用到 action，如果没有省略了

`mapDispatchProps` ，组件要触发 action 就必须手动 dispatch，类似这样： `this.props.dispatch(someActionCreator('arg'))` 。

表单

表单不同于其他 HTML 元素，因为它要响应用户的交互，显示不同的状态，所以在 React 里面会有点特殊。

状态属性

表单元素有这么几种属于状态的属性：

- `value` ，对应 `<input>` 和 `<textarea>` 所有
- `checked` ，对应类型为 `checkbox` 和 `radio` 的 `<input>` 所有
- `selected` ，对应 `<option>` 所有

在 *HTML* 中 *`<textarea>`* 的值可以由子节点（文本）赋值，但是在 *React* 中，要用 *`value`* 来设置。

表单元素包含以上任意一种状态属性都支持 `onChange` 事件监听状态值的更改。

针对这些状态属性不同的处理策略，表单元素在 React 里面有两种表现形式。

受控组件

对于设置了上面提到的对应“状态属性“值的表单元素就是受控表单组件，比如：

```
render: function() {
    return <input type="text" value="hello"/>;
}
```

一个受控的表单组件，它所有状态属性更改涉及 UI 的变更都由 React 来控制（状态属性绑定 UI）。比如上面代码里的 `<input>` 输入框，用户输入内容，用户输入的内容不会显示（输入框总是显示状态属性 `value` 的值 `hello` ），这有点颠覆我们的认知了，所以说这是受控组件，不是原来默认的表单元素了。

如果你希望输入的内容反馈到输入框，就要用 `onChange` 事件改变状态属性 `value` 的值：

```
getInitialState: function() {
    return {value: 'hello'};
},
handleChange: function(event) {
    this.setState({value: event.target.value});
},
render: function() {
    var value = this.state.value;
    return <input type="text" value={value} onChange={this.handleCh
}
```

使用这种模式非常容易实现类似对用户输入的验证，或者对用户交互做额外的处理，比如截断最多输入140个字符：

```
handleChange: function(event) {
    this.setState({value: event.target.value.substr(0, 140)});
}
```

非受控组件

和受控组件相对，如果表单元素没有设置自己的“状态属性”，或者属性值设置为 `null` ，这时候就是非受控组件。

它的表现就符合普通的表单元素，正常响应用户的操作。

同样，你也可以绑定 `onChange` 事件处理交互。

如果你想要给“状态属性”设置默认值，就要用 React 提供的特殊属性 `defaultValue` ，对于 `checked` 会有 `defaultChecked` ， `<option>` 也是使用 `defaultValue` 。

为什么要有受控组件？

引入受控组件不是说它有什么好处，而是因为 React 的 UI 渲染机制，对于表单元素不得不引入这一特殊的处理方式。

在浏览器 DOM 里面是有区分 *attribute* 和 *property* 的。*attribute* 是在 HTML 里指定的属性，而每个 HTML 元素在 JS 对应是一个 DOM 节点对象，这个对象拥有的属性就是 *property*（可以在 console 里展开一个 DOM 节点对象看一下，HTML *attributes* 只是对应其中的一部分属性），*attribute* 对应的 *property* 会从 *attribute* 拿到初始值，有些会有相同的名称，但是有些名称会不一样，比如 *attribute* `class` 对应的 *property* 就是 `className`。（详细解释：.prop，.prop() vs .attr()）

回到 React 里的 `<input>` 输入框，当用户输入内容的时候，输入框的 `value` *property* 会改变，但是 `value` *attribute* 依然会是 HTML 上指定的值（*attribute* 要用 `setAttribute` 去更改）。

React 组件必须呈现这个组件的状态视图，这个视图 HTML 是由 `render` 生成，所以对于

```
render: function() {
    return <input type="text" value="hello"/>;
}
```

在任意时刻，这个视图总是返回一个显示 `hello` 的输入框。

`<select>`

在 HTML 中 `<select>` 标签指定选中项都是通过对应 `<option>` 的 `selected` 属性来做的，但是在 React 修改成统一使用 `value`。

所以没有一个 **`selected`** 的状态属性。

```
<select value="B">
    <option value="A">Apple</option>
    <option value="B">Banana</option>
    <option value="C">Cranberry</option>
</select>
```

你可以通过传递一个数组指定多个选中项：`<select multiple={true} value={['B', 'C']}>`

服务器端渲染

React 提供了两个方法 `renderToString` 和 `renderToStaticMarkup` 用来将组件（Virtual DOM）输出成 HTML 字符串，这是 React 服务器端渲染的基础，它移除了服务器端对于浏览器环境的依赖，所以让服务器端渲染变成了一件有吸引力的事情。

服务器端渲染除了要解决对浏览器环境的依赖，还要解决两个问题：

- 前后端可以共享代码
- 前后端路由可以统一处理

React 生态提供了很多选择方案，这里我们选用 Redux 和 react-router 来做说明。

Redux

Redux 提供了一套类似 Flux 的单向数据流，整个应用只维护一个 Store，以及面向函数式的特性让它对服务器端渲染支持很友好。

2 分钟了解 Redux 是如何运作的

关于 Store：

- 整个应用只有一个唯一的 Store
- Store 对应的状态树（State），由调用一个 reducer 函数（root reducer）生成
- 状态树上的每个字段都可以进一步由不同的 reducer 函数生成
- Store 包含了几个方法比如 `dispatch`, `getState` 来处理数据流
- Store 的状态树只能由 `dispatch(action)` 来触发更改

Redux 的数据流：

- action 是一个包含 `{ type, payload }` 的对象
- reducer 函数通过 `store.dispatch(action)` 触发
- reducer 函数接受 `(state, action)` 两个参数，返回一个新的 state
- reducer 函数判断 `action.type` 然后处理对应的 `action.payload` 数据来更新状态树

所以对于整个应用来说，一个 Store 就对应一个 UI 快照，服务器端渲染就简化成了在服务器端初始化 Store，将 Store 传入应用的根组件，针对根组件调用 `renderToString` 就将整个应用输出成包含了初始化数据的 HTML。

react-router

react-router 通过一种声明式的方式匹配不同路由决定在页面上展示不同的组件，并且通过 props 将路由信息传递给组件使用，所以只要路由变更，props 就会变化，触发组件 re-render。

假设有一个很简单的应用，只有两个页面，一个列表页 `/list` 和一个详情页 `/item/:id`，点击列表上的条目进入详情页。

可以这样定义路由，`./routes.js`

```
import React from 'react';
import { Route } from 'react-router';
import { List, Item } from './components';

// 无状态（stateless）组件，一个简单的容器，react-router 会根据 route
// 规则匹配到的组件作为 `props.children` 传入
const Container = (props) => {
  return (
    <div>{props.children}</div>
  );
};

// route 规则：
// - `/list` 显示 `List` 组件
// - `/item/:id` 显示 `Item` 组件
const routes = (
  <Route path="/" component={Container} >
    <Route path="list" component={List} />
    <Route path="item/:id" component={Item} />
  </Route>
);

export default routes;
```

从这里开始，我们通过这个非常简单的应用来解释实现服务器端渲染前后端涉及的一些细节问题。

Reducer

Store 是由 reducer 产生的，所以 reducer 实际上反映了 Store 的状态树结构

`./reducers/index.js`

```
import listReducer from './list';
import itemReducer from './item';

export default function rootReducer(state = {}, action) {
  return {
    list: listReducer(state.list, action),
    item: itemReducer(state.item, action)
  };
}
```

`rootReducer` 的 `state` 参数就是整个 Store 的状态树，状态树下的每个字段对应也可以有自己的 reducer，所以这里引入了 `listReducer` 和 `itemReducer` ，可以看到这两个 reducer 的 state 参数就只是整个状态树上对应的 `list` 和 `item` 字段。

具体到 `./reducers/list.js`

```
const initialState = [];

export default function listReducer(state = initialState, action) {
  switch(action.type) {
  case 'FETCH_LIST_SUCCESS': return [...action.payload];
  default: return state;
  }
}
```

list 就是一个包含 items 的简单数组，可能类似这种结构： `[{ id: 0, name: 'first item'}, {id: 1, name: 'second item'}]` ，从 `'FETCH_LIST_SUCCESS'` 的 `action.payload` 获得。

然后是 `./reducers/item.js` ，处理获取到的 item 数据

```
const initialState = {};

export default function listReducer(state = initialState, action) {
  switch(action.type) {
  case 'FETCH_ITEM_SUCCESS': return [...action.payload];
  default: return state;
  }
}
```

Action

对应的应该要有两个 action 来获取 list 和 item，触发 reducer 更改 Store，这里我们定义 `fetchList` 和 `fetchItem` 两个 action。

`./actions/index.js`

```
import fetch from 'isomorphic-fetch';

export function fetchList() {
  return (dispatch) => {
    return fetch('/api/list')
        .then(res => res.json())
        .then(json => dispatch({ type: 'FETCH_LIST_SUCCESS', payloa
  }
}

export function fetchItem(id) {
  return (dispatch) => {
    if (!id) return Promise.resolve();
    return fetch(`/api/item/${id}`)
        .then(res => res.json())
        .then(json => dispatch({ type: 'FETCH_ITEM_SUCCESS', payloa
  }
}
```

isomorphic-fetch 是一个前后端通用的 Ajax 实现，前后端要共享代码这点很重要。

另外因为涉及到异步请求，这里的 action 用到了 thunk，也就是函数，redux 通过 `thunk-middleware` 来处理这类 action，把函数当作普通的 action dispatch 就好了，比如 `dispatch(fetchList())`

Store

我们用一个独立的 `./store.js`，配置（比如 Apply Middleware）生成 Store

```
import { createStore } from 'redux';
import rootReducer from './reducers';

// Apply middleware here
// ...

export default function configureStore(initialState) {
  const store = createStore(rootReducer, initialState);
  return store;
}
```

react-redux

接下来就是实现 `<List>` ， `<Item>` 组件，然后把 Redux 和 React 组件关联起来，具体细节参见 react-redux

`./app.js`

```
import React from 'react';
import { render } from 'react-dom';
import { Router } from 'react-router';
import createBrowserHistory from 'history/lib/createBrowserHistory
import { Provider } from 'react-redux';
import routes from './routes';
import configureStore from './store';

// `__INITIAL_STATE__` 来自服务器端渲染，下一部分细说
const initialState = window.__INITIAL_STATE__;
const store = configureStore(initialState);
const Root = (props) => {
  return (
    <div>
      <Provider store={store}>
        <Router history={createBrowserHistory()}>
          {routes}
        </Router>
      </Provider>
    </div>
  );
}

render(<Root />, document.getElementById('root'));
```

至此，客户端部分结束。

Server Rendering

接下来的服务器端就比较简单了，获取数据可以调用 action，routes 在服务器端的处理参考 react-router server rendering，在服务器端用一个 `match` 方法将拿到的 request url 匹配到我们之前定义的 routes，解析成和客户端一致的 props 对象传递给组件。

`./server.js`

```
import express from 'express';
```

```
import React from 'react';
import { renderToString } from 'react-dom/server';
import { RoutingContext, match } from 'react-router';
import { Provider } from 'react-redux';
import routes from './routes';
import configureStore from './store';

const app = express();

function renderFullPage(html, initialState) {
  return `
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
    </head>
    <body>
      <div id="root">
        <div>
          ${html}
        </div>
      </div>
      <script>
        window.__INITIAL_STATE__ = ${JSON.stringify(initialState)};
      </script>
      <script src="/static/bundle.js"></script>
    </body>
    </html>
  `;
}

app.use((req, res) => {
  match({ routes, location: req.url }, (err, redirectLocation, renc
    if (err) {
      res.status(500).end(`Internal Server Error ${err}`);
    } else if (redirectLocation) {
      res.redirect(redirectLocation.pathname + redirectLocation.sea
    } else if (renderProps) {
      const store = configureStore();
      const state = store.getState();
```

```
        Promise.all([
          store.dispatch(fetchList()),
          store.dispatch(fetchItem(renderProps.params.id))
        ])
        .then(() => {
          const html = renderToString(
            <Provider store={store}>
              <RoutingContext {...renderProps} />
            </Provider>
          );
          res.end(renderFullPage(html, store.getState()));
        });
      } else {
        res.status(404).end('Not found');
      }
    });
  });
```

服务器端渲染部分可以直接通过共用客户端 `store.dispatch(action)` 来统一获取 Store 数据。另外注意 `renderFullPage` 生成的页面 HTML 在 React 组件 mount 的部分(`<div id="root">`)，前后端的 HTML 结构应该是一致的。然后要把 `store` 的状态树写入一个全局变量（ `__INITIAL_STATE__` ），这样客户端初始化 render 的时候能够校验服务器生成的 HTML 结构，并且同步到初始化状态，然后整个页面被客户端接管。

最后关于页面内链接跳转如何处理？

react-router 提供了一个 `<Link>` 组件用来替代 `<a>` 标签，它负责管理浏览器 history，从而不是每次点击链接都去请求服务器，然后可以通过绑定 `onClick` 事件来作其他处理。

比如在 `/list` 页面，对于每一个 item 都会用 `<Link>` 绑定一个 route url： `/item/:id` ，并且绑定 `onClick` 去触发 `dispatch(fetchItem(id))` 获取数据，显示详情页内容。

更多参考

- Universal (Isomorphic)
- isomorphic-redux-app

[最初发布于 Coding Blog]